Frontend with JavaScript

Making Requests from HTML Pages to PHP APIs

How does a Web Application work

- Users request a web page through a web browser.
    - The web server (e.g., Nginx, Apache) receives the request and routes it to the appropriate server-side program (e.g., PHP).
    - The PHP server processes the request and generates a response, usually in the form of HTML, CSS, and JavaScript.
    - The web browser receives this response and renders the page for the user.
- Once the page is loaded, users interact with the front end (JavaScript).

- The front end (JavaScript) can send additional requests (often via AJAX or Fetch API) to the server (e.g., for APIs or dynamic content).
- The same PHP backend or other API endpoints usually handle these requests.

```
[HTML + JavaScript] ⟷ HTTP Requests ⟷ [PHP Server]
     (Frontend)                          (Backend)
```

NOTICE!

- For simplicity, we have taken a basic approach to understand web applications:
  - We have created and run the frontend directly on the client side (e.g., using local HTML/JS files), rather than having it served by the PHP backend.
  - We have used the built-in PHP development server (php -S), instead of a full web server like NGINX or Apache.
- Later, we will:
  - Use a real web server (e.g., NGINX or Apache)
  - Write PHP code to serve the frontend files, creating a complete and realistic web application setup.

Web Application Frontend Tools We'll Use:

- HTML for structure
- CSS for styling
- JavaScript for API communication
- Fetch API for HTTP requests

What We're Building for Web Applications

1. **HTML Page**: User interface with buttons and forms
2. **JavaScript**: Makes HTTP requests to your PHP API
3. **PHP API**: Processes requests and returns JSON
4. **Dynamic Updates**: Page updates without refreshing

Example: Communication between Frontend and Backend

Frontend (`test.html`) ⟷ **Backend** (`index.php`)

Backend Code: index.php

- To access this server, users should make a GET request.
  - http://localhost:3000/index.php/api?a=b&c=d
- Backend (PHP) extracts the API information (api) from this GET request.

```php
// Get the request path and clean it up
$path = $_SERVER['REQUEST_URI']; // '/index.php/api?a=b&c=d'
// remove ?a=b&c=d
$path = parse_url($path, PHP_URL_PATH); // '/index.php/api'
$path = trim($path, '/'); // 'index.php/api'

// Remove index.php from path if present
if (strpos($path, 'index.php') === 0) { // true as path has index.php
    $path = substr($path, 9); // '/api'
    $path = trim($path, '/'); // 'api'
}
```

Making a Response

- In this example, the PHP server returns JSON as a response.
    - It generates multiple information, such as message, data, and count.

```php
<?php
function sendResponse($data, $message = 'Success') {
    echo json_encode([
        'success' => true,
        'message' => $message,
        'data' => $data,
        'count' => is_array($data) ? count($data) : 1
    ], JSON_PRETTY_PRINT);
}
```

Making an Error Response (400)

- Following the web protocol, we send a 400 error when an error occurs.

```php
function sendError($message, $code = 400) {
    http_response_code($code);
    echo json_encode([
        'success' => false,
        'message' => $message,
        'data' => null
    ], JSON_PRETTY_PRINT);
}
```

Routing the $path to handle the API

- In this example, we handle only the `api` API.

```
switch ($path) {
    case 'api':
        // API information endpoint
        $info = [
            'name' => 'Simple Student Management API',
            'version' => '1.0',
            'description' => 'A minimal API for learning PHP basics with student data',
            'endpoints' => [
                'GET /api' => 'Show this API information',
            ]
        ];
        sendResponse($info, 'Welcome to Simple Student Management API');
        break;
    default:
        sendError('Endpoint not found', 404);
        break;
}
?>
```

Frontend Code: test.html

- To access the API, we need to make a request.
  - Run PHP server with <PHP -S localhost:8000>
  - http://localhost:8000/index.php/api
- This is a response from the server.


```
{
    "success": true,
    "message": "Welcome to Simple Student Management API",
    "data": {
        "name": "Simple Student Management API",
        "version": "1.0",
        "description": "A minimal API for learning PHP basics with student data",
        "endpoints": {
            "GET \/api": "Show this API information"
        }
    },
    "count": 4
}
```

HTML/JavaScript

- We need to get the response using HTML/JavaScript.

```
<!DOCTYPE html>
<html lang="en">
<head>
    ...
</head>
<body>
    <div class="container">
        <!-- Button to trigger API call -->
        <button onclick="getApiInfo()">Get API Information</button>
        <!-- Area to display results -->
        <div id="result" class="result">
            Click any button above to test the API...
        </div>
    </div>
    <script>/* JavaScript code */</script>
</body>
</html>
```

JavaScript: The Communication Layer

- We use the fetch JavaScript API to make a request.

```
// Simple GET request
fetch('/index.php/api')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));
```

- We can use async/await to get the same results (preferred)

```
async function getApiInfo() {
    try {
        const response = await fetch('/index.php/api');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}
```

Why async/await? Cleaner, easier to read, better error handling

Displaying the response from servers

```
// Base URL for our PHP API
const API_BASE = '/index.php';

// Function to make API calls
async function apiCall(endpoint) {
    try {
        // Make the request
        const response = await fetch(API_BASE + endpoint);
        const data = await response.json();

        // Display the result
        document.getElementById('result').textContent =
            JSON.stringify(data, null, 2);

    } catch (error) {
        console.error('Error:', error);
    }
}
```

- We make the api GET request using this `getApiInfo()` JavaScript function.

```
function getApiInfo() {
   apiCall('/api');  // Calls our API at /index.php/
}
```

Button Click Handlers

- HTML Button

```
<button onclick="getApiInfo()">Get API Information</button>
```

- JavaScript Handler

```
function getApiInfo() {
    apiCall('/api');  // Calls our API at /index.php/
}
```

What Happens

1. User clicks the button
2. `getApiInfo()` function runs
3. `apiCall('/api')` makes an HTTP request to `/index.php/api`
4. PHP processes the request and returns JSON
5. JavaScript receives the response and updates the page

Different Types of Requests

- We can make other types of requests using JavaScript.
    - We should make corresponding API handlers on the PHP side.

- GET Request (Default)

```
const response = await fetch('/api/users');
```

- POST Request (JSON)

```
const response = await fetch('/api/users', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        name: 'John Doe',
        email: 'john@example.com'
    })
});
```

- PUT Request (JSON)

```
const response = await fetch('/api/users/123', {
    method: 'PUT',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        name: 'John Smith'
    })
});
```

Error Handling Best Practices

- Check Response Status

```
async function apiCall(endpoint) {
    try {
        const response = await fetch(API_BASE + endpoint);

        // Check if request was successful
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`);
        }

        const data = await response.json();
        return data;

    } catch (error) {
        console.error('API call failed:', error);
        throw error; // Re-throw for caller to handle
    }
}
```

Handle Different Error Types

```
try {
    const data = await apiCall('/api/users');
    // Success handling
} catch (error) {
    if (error.name === 'TypeError') {
        // Network error
        showError('Network error. Please check your connection.');
    } else {
        // Server error
        showError(`Server error: ${error.message}`);
    }
}
```

Updating the DOM with Results

- Simple Text Update

```
document.getElementById('result').textContent = 'Hello World';
```

- CSS/HTML

```
<style>
  #result {
    background: #f5f5f5; padding: 1em; border: 1px solid #ccc;
    font-family: monospace; white-space: pre-wrap; word-break: break-word;
  }
</style>
```

```
<div id="result" class="result">
  Click any button above to test the API...
</div>
```

- HTML Update from the PHP server

```
const response = await fetch(API_BASE + endpoint);
const data = await response.json();
// Display the result
document.getElementById('result').textContent =
  JSON.stringify(data, null, 2);
```

Key Takeaways

- Frontend-Backend Communication:

1. **HTML** provides the user interface structure
2. **JavaScript** handles API communication and DOM updates
3. **Fetch API** makes modern HTTP requests easy
4. **async/await** provides clean asynchronous code
5. **Error handling** is crucial for a good user experience

Best Practices

- Always handle errors gracefully
- Provide user feedback (loading states)
- Use browser developer tools for debugging
- Keep API calls focused and straightforward